NLP2 Project 3 description

Miloš Stanojević, Philip Schülz and Wilker Aziz

May 13, 2016

This time your task is to make the best possible prediction (a translation) from a list of candidate derivations (n-best derivations from a phrase-based model).

A common approach is to learn a ranking model that tries (explicitly or not) to correlate model score to some form of oracle score. In this case, the oracle is an automatic evaluation metric (e.g. BLEU, METEOR, BEER). Your supervision is the set of reference translations and your space of translation options is constrained to what is available in 1000-best lists.

In general terms you will work with

- design/induction of rich features that describe the translation derivations
- learning to rank
- automatic evaluation

1 Features

Some of the potential features are listed bellow. Note that the choices of features you are using might influence the learning algorithm implementation (for example sparse vs. dense vectors) and it also might require extra tools (in case of syntactic trees it would require a parser).

You should use/implement at least one feature from each of the following subsections: 1.1, 1.2, 1.3 and 1.4.

1.1 Core Features

Some standard features (language model, translation model, inverse translation model) are already precomputed for you and available in your n-best list file. You should probably use those since they are very informative.

1.2 Feature Combinations

You can get additional features by combining core features with some non-linear function. For example, if you have two core features x_1 and x_2 you can create

new features that are computed as x_1^2, $x_1 * x_2$, x_2^2, $sigmoid(x_1)$... or anything you think would be useful.

1.3 Linguistically Inspired Features

You can use some of your linguistic knowledge in designing features. For example, most of the sentences have a verb, so you might like to have a binary feature that says if there is a verb in a sentence. Naturally for this you need some parser.

1.4 Sparse Features of Bigrams

As you know, bigrams are two consecutive words in some sentence. Here we will generalize this concept and say that bigrams can be two consecutive POS tags or two consecutive cluster IDs. Why would we want to do that? Sometimes our training data is too small and we will not observe majority of potential word bigrams. Because of that we need to abstract to some more general category than words such as POS tags. So instead of collecting statistics about bigrams like "bird spoke", "cow sang"... we would collect statistics about POS bigram "Noun Verb".

If you use bigram features, each bigram will be one indicator feature. So if you had observed in the candidate sentence a bigram "Preposition Verb" then that feature would have value 1 otherwise it will be 0. You can already imagine that majority of these features will be 0 and for that you will want to encode them as *sparse features*. That may influence the way you implement the tuning algorithm.

These sparse features of bigram POS tags don't have to be extracted from the linear order of words in the sentence. You can use a dependency parse tree, for example, and there bigram would be POS tags connected by a dependency arc.

If you use the syntacic trees then for German you can consider using this parser `https://code.google.com/archive/p/mate-tools/wikis/ParserAndModels.wiki` (with it you will also get POS tags for free).

If you want to use word clusters then this word clustering toolkit might help out `http://nlp.stanford.edu/software/tagger.shtml`.

1.5 Extra: Feature Induction

Instead of designing by hand it is often more efficient to induce them from data. This has two main advantages. First, there is no need to determine the usefulness of features through trial and error (assuming that your feature induction mechanism works well). Second, induced features tend to be dense and thus are much easier to use for down-stream purposes than sparse features.

We ask you to induce word representations. These may be combined into phrase representations by simple addition or any other procedure that you can come up with. Standard software like `Word2Vec` induces features monolingually.

Here, you will have to induce features for target words using conditional on the source sentence. To this end you will use restricted Boltzmann Machines (see [1]. A RBM is a generative neural network with binary hidden states. Each hidden state is fully connected with each data point (target word). The weights that are learned by the RBM can be used as representations. How RBMs can be learned efficiently on word observation is described in [2]. Using the RBM, you will train target word representations that are influence by the source sentence. This can, for example, be achieved by additionally connecting the hidden units to all words in the source sentence or by learning a representation for the source sentence and connecting the hidden units to this representation. The concrete architecture is up to your imagination but please make sure to consult us before you actually start implementing it.[1]

2 Implementing the Tuning Algorithm

After you decide on your feature sets you will need a learning algorithm that would learn weights for these features. In this project, it is obligatory to implement the learning algorithm PRO for linear models. After describing a bit of PRO that you need I will mention some extra things you might want to play with such as different learning algorithm for a linear model or even having a non-linear model.

2.1 PRO

PRO is probably the most simple algorithm that could be used for tuning. The main idea of it is to transform tuning task into a binary classification task. That is done by: 1) selecting pairs of candidate translations from n-best lists, 2) transforming pairs of training translations into training instances for a linear classifier, 3) train any off-the-shelf linear classifier, 4) return the weights learned from the classifier.

The number of pairs of translations that need to be extracted depends on you. Usually, the more the better, but you can't have all the pairs because there is huge number of them. That's why you will need to randomly pick them.

Transforming those pairs into training instances is trivial. For positive training instance you subtract features of better translation and worse translation in each pair of translations. For negative training instance you do it in the opposite direction (subtract features of worse and better).

You do not have to implement a classifier yourself (unless you really want to, but then be careful not to lose track of the project's timeline). There are many linear classifiers that you can use for training your model. The fastest and maybe the simplest in my experience is LibLinear `https://www.csie.ntu.edu.tw/~cjlin/liblinear/`, but you can really use anything. For example, for python you can use ScikitLearn `http://scikit-learn.org/stable/` (scipy also offers a few optimisation algorithms that may be useful).

[1] If you are interested in RBMs, Philip volunteered to discuss these ideas further.

The pseudo code of this algorithm is presented in Algorithm 1. This is a simplified version of PRO. The original version was published in [3], but that one had a bit more hacks to it that you don't need to implement (mostly about techniques for sampling).

Algorithm 1 PRO pseudo code

```
Input1: nbests – n-best for each sentence
Input2: sampleSize
Output: w – weight vector
classifierTrainingInstances ← {}
for all nbest in nbests do
  for i ← 1 to sampleSize do
    candidate1 ← randomly pick from nbest
    candidate2 ← randomly pick from nbest
    winner ← better from candidate1 and candidate2 as judged by BLEU
    loser ← worse from candidate1 and candidate2 as judged by BLEU
    winFeatures ← featureExtraction(winner)
    losFeatures ← featureExtraction(loser)
    posInstance ← <features = winFeatures–losFeatures, label=1>
    negInstance ← <features = losFeatures–winFeatures, label=-1>
    classifierTrainingInstances.add(posInstance)
    classifierTrainingInstances.add(negInstance)
  end for
end for
train off the shelf linear classifier with classifierTrainingInstances
return weights learned by the classifier
```

2.2 Extra

2.2.1 MIRA

You might want to implement MIRA learning algorithm. Just like PRO it produces a linear model, but unlike PRO it is an online training algorithm and you can't use arbitrary classifier to learn the weights. You would need to do several iterations in which you would find the *hope* and *fear* translations in nbest lists and use MIRA update rule to find the new weights. In Algorithm 2 you can find a very rough idea of how it works. More precise description (particularily for $updateParameters$) can be found here [4].

2.2.2 Non-linear reranking

The main reason why we use linear models very often in machine translation is that they decompose well while we do decoding. Since you are not learning these weights for decoding but for reranking n-best lists you don't have to stay with linear models. There are many learning algorithms that can train

Algorithm 2 MIRA pseudo code

Input1: nbests – n-best for each sentence
Input2: numberOfIterations
Output: **w** – weight vector
$\boldsymbol{w} \leftarrow zerovector$
for iteration $\leftarrow 1$ **to** numberOfIterations **do**
  for all nbest in nbests **do**
    $hope \leftarrow argmax_{t \in nbest} \boldsymbol{w} * features(t) + BLEU(hopeTranslation)$
    $fear \leftarrow argmax_{t \in nbest} \boldsymbol{w} * features(t) - BLEU(hopeTranslation)$
    $updateParameters(\mathbf{w}, hope, fear)$
  end for
end for

non-linear models for better ranking. However, very often they are difficult to implement and they are very slow. If you want, you can give a try to some algorithms that are already implemented in some learning-to-rank toolkit such as RankLib `https://people.cs.umass.edu/~vdang/ranklib.html`. If you know any other library that you like more or know how to do anything that would make reranking approach be better feel free to do it.

3 Evaluation Metrics

In the explanation of this project we usually mention BLEU evaluation metric, but you do not have to use BLEU. Some other metrics might give better result in tuning.

The tuning algorithm that you are implementing PRO does evaluation on the sentence level and BLEU is very often unreliable on the sentence level. You can guess that the reason for that is because high order n-grams (4-gram for example) often don't match and they cause the whole sentence to have score 0. To avoid that you can do some smoothing of the BLEU score by adding a fake count 1 to each n-gram order. So basically instead of computing precision with

$$\frac{\#matching\_ngrams}{\#total\_ngrams} \tag{1}$$

you will compute it with

$$\frac{1 + \#matching\_ngrams}{1 + \#total\_ngrams} \tag{2}$$

In that way BLEU will never be 0.

Alternatively, you can try some completely different metric. For example, you can try METEOR `http://www.cs.cmu.edu/~alavie/METEOR/`. You should know that METEOR usually gives much more importance to recall than precision so after tuning your parameters might prefer too long translations. To avoid that with METEOR you need to set equal weight for precision and recall.

4 Data

You will be working with English-German translations. We are providing 1000-best lists already ranked from best to worst by a phrase-based decoder (Moses) using the standard translation features and a 5-gram language model. The baseline system translates lattices containing up to 100 permutations of English sentences using the preordering method of [5]. The baseline features are:

- a German 5-gram language model trained on monolingual news-2015 data (from WMT – see below)
- permutation distortion features – this feature captures the quality of the input permutations
- a preordered-English 5-gram language model trained on the (preordered) English side of the English-German europarl (from WMT)
- expected Kendall Tau score under the reordering grammar of [5] – this feature captures the quality of the input permutations
- unknown word penalty
- word penalty
- phrase penalty
- translation probabilities (phrase and lexical in both directions)
- something called `InputFeature` which is a dummy feature that can be ignored (it is 0 everywhere, so it does not affect the model)

In case you need training data for additional features/models, you can find it (both parallel and monolingual) in the website of the WMT annual shared task: `http://www.statmt.org/wmt16/`.

4.1 Files

You can download the data from:

`https://www.adrive.com/public/Kyuzep/nlp2`

You will find the following files:

- `nlp2-dev.1000best.gz`: 1000-best lists for 2937 dev sentences.
- `nlp2-test.1000best.gz`: 1000-best lists for 2107 test sentences.
- `baseline.weights`: weights used in producing the n-best lists
- `nlp2-dev.en.pw.plf-100.gz`: input lattice for the dev set

- `nlp2-test.en.pw.plf-100.gz`: input lattice for test set.
  You are unlikely to need the input lattices, but we attach them just in case.[2]
- `nlp2-dev.de.gz` and `nlp2-test.de.gz`: reference translations
- `nlp2-dev.en.s.gz` and `nlp2-test.en.s.gz`: input sentences (in original word order).

The n-best lists provided are formatted as columns separated by 3 vertical bars. The columns contain the following information:[3]

1. 0-based sentence id
2. translation segmented into phrases: phrase boundaries are denoted with $|a - b|$ where a and b are states in the input lattice
3. core feature vector: baseline features used to produced the n-best lists
4. system score: dot product between core features and baseline weight vector
5. alignment: $i - j$ pairs, where i represents a state in the input lattice and j represents a target word position (0-based)
6. source sentence: the source words along the exact path in the input lattice that got translated.

5 Final Evaluation

After the learning phase is finished you will have your reranking model that you should apply on the test data. After reranking the test set you will take the top scored translations in nbest lists as your final translations. When this corpora of translations is constructed you will evaluate it with several evaluation metrics.

For that you can use a very useful toolkit called MultEval `https://github.com/jhclark/multeval`. It will produce BLEU, METEOR and TER scores. Very nice thing about MultEval is that it can compute statistical significance scores for the metrics scores so you would know if the improvement that you got is significantly better than some baseline. As a baseline you will you the weight that you will get from us.

In your report you will show results for:

- baseline – just standard weights without your training

[2] You can find a python script for parsing such PLF files at `https://github.com/wilkeraziz/grasp/blob/master/grasp/io/plf_parser.py`. You don't need to download the whole toolkit, just the standalone file`plf_parser.py`. PLF format is described in `http://www.statmt.org/moses/?n=Moses.WordLattices`.

[3] For more information you can check Moses's documentation: `http://www.statmt.org/moses`

- basic – system tuned with basic features and your impelemntation of PRO
- anything else that you impelemnt

In your results you should report ablation experiments – for everything you add to your model you should show how much it added (or decreased) the scores.

6 Report

Again, we expect you to write about your research and empirical findings in the form of a research paper. A few things we expect to find in your report:

- explain and motivate your choice of features
  - if you induced features, give an overview of the method used (again, always motivate your choices)
- explain the training algorithms you implemented, use the following (where applicable) as a guidelines
  - choice of learning objective
  - choice of regularisation method
  - choice of classifier
  - choice of sampling procedure (e.g. when creating negative examples for pairwise classification)
  - choice of hyperparameters (e.g. number of negative samples, or other hyperparameters associated with your choice of machine learning algorithm)
- be critical about findings

You will be assessed mainly in terms of your report, but do submit a link to your code base and output translation files. In grading your project, we will take into account the following guidelines/criteria:

- theoretical description (4 points): did you convey a concrete and correct understanding of the methodology?
- empirical evaluation (4 points): did you succeed in the proposed empirical investigation? Is evaluation done in a proper way
- writing style (2 points)

References

[1] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Compututation*, 18(7):1527–1554, 2006.

[2] George E. Dahl, Ryan P. Adams, and Hugo Larochelle. Training restricted Boltzmann machines on word observations. In *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, ICML '12, pages 679–686, 2012.

[3] Mark Hopkins and Jonathan May. Tuning as ranking. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, pages 1352–1362, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[4] Vladimir Eidelman. Optimization strategies for online large-margin learning in machine translation. In *Proceedings of the Seventh Workshop on Statistical Machine Translation*, WMT '12, pages 480–489, Stroudsburg, PA, USA, 2012. Association for Computational Linguistics.

[5] Miloš Stanojević and Khalil Sima'an. Reordering grammar induction. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 44–54, Lisbon, Portugal, September 2015. Association for Computational Linguistics.